University of Illinois at Urbana-Champaign
Department of Computer Science

MP4: Lists – Interface and Implementation

CS 225 Data Structures
Spring Semester, 1999

Handed out: Saturday, February 20, 1999
Due date: Saturday, February 27, 1999 at 5:00 PM

1 Introduction

In this MP, you will gain experience with both the `List` ADT and the linked implementation of lists. At this point, it is assumed that you are familiar enough with C++ to be able to worry more about the logic flow of a program, rather than focusing only on the syntax. In addition, the code that is given is all code you have seen before – we have already used the `Array` class in earlier MPs, and you spent last week's section going over the singly-linked `List` class.

2 The programming assignment

To begin with, you will need to copy the given files into your own directory. The files are located in:

```
~cs225/src/mp4          (MP code)
~cs225/src/mp4/test     (test files)
```

3 Adding to the `List` class

The `singlist.*` files contain the code you looked at in the previous discussion section. However, you will notice that at the end of `singlist.h`, there are five more function headers that were not there before ("before" meaning, before this MP was released, i.e. five function headers that do not appear in the library copy of the code and were never discussed in section).

The specification of these five functions is as follows:

- `Replace` – this function takes two parameters, both of the generic type. This function will search the list for every occurence of the first parameter's value, and wherever that value is found, it will be replaced with the second parameter's value.
- `ToArray`. This function takes no parameters, but will return an `Array` object whose size is equal to the size of the list (and whose lowest index is 1). This function merely returns an `Array` that stores the elements of the list in the same order in which they appear in the list itself. This function assumes that the list is non-empty (since our `Array` class is not set up to handle arrays of zero cells) and will not necessarily work correctly on empty lists.
- `SortedInsert` – this function takes a single parameter of the generic type. The function requires two things before it can be called: first, that the "less than" operator (`<`) must be defined on whatever real type you substitute for the generic type, and second, that the

list must currently be sorted from smallest value to largest value (as defined by the "less than" operator). The function will not compile if the first requirement does not hold, and it assumes the second requirement holds, meaning that the user must not call this function on an unsorted list.

Assuming the requirements hold, the value of generic type will be inserted into the list in sorted order. If this value already occurs in the list, then the new occurence of it should be inserted after all occurences already there. For example, if you are inserting "5" into a list of integers, and there are already two "5"s in the list, then the new "5" should be inserted as the third "5" in the list, not as the first or second "5". At the end of this function, the `current` pointer should point to the newly inserted value.

- `Reverse` – this function, which takes no parameters, simply reverses the order of the elements in the list.

- `Splice` – this function takes a single parameter, a second list. What the function will do is splice the entire second list into the first list *after* the `current` value of the first list. For example, if your first list (the object you actually call the function off of) is `(3 7 8 2 1)` with the value 8 being the `current` value, and if the parameter list is `(4 9 0)`, then this function results in the original list now being `3 7 8 4 9 0 2 1)`. The second list should be empty once you are done – that is, you are not splicing a *copy* of the second list into the first; rather, you are splicing the *actual* second list into the first, leaving the second list empty.

Your first task is to write the definitions for these five member functions. They should be added to the file `singlist.C`. However, there are a few things you must keep in mind when writing these definitions:

1. Except for `SortedInsert`, which assigns `current` to refer to the newly-inserted element, all the functions should assign `current` to refer to the first element in the list (or `NULL` if the list is empty) when the function has completed.

2. Please write your five function definitions *before* the `List` constructors, as indicated in the given `singlist.C` file. Normally you would list the functions in the `.C` in the same order they were listed in the interface, simply to make each function easy to find. But, in this case, we want the functions to be easy to find for the *graders*, and so it is helpful to have your code all in one place at the start of the file.

3. You are **NOT!!!!!!!!** allowed to call any of the `List` functions **anywhere** in your implementations for the five functions above, and if you do so, we reserve the right to grade that function and its output however we see fit. You *are* allowed to call the `ListNode` constructor, and of course you can declare pointers to `ListNode` objects. In fact, simply on the basis of needing to write an insertion function, it should be clear that you will need to do both of those things. But, you cannot call any of the `List` functions themselves. This includes the `List` constructors, which means that your functions should not be declaring any extra `List` objects to manipulate. All manipulation of the nodes themselves must be done using the fields of the node and not functions calls to pre-written `List` functions – though you can of course draw inspiration from any of the code for the `List` functions as you see fit.

4 Using the `List` interface

For the second part of your MP, you are going to re-implement the same functions, but this time, you are going to do it from the point of view of the user. That is, rather than change the actual implementation of the class itself, you will instead use the `List` interface to operate on `List` objects and you will code the requested algorithms that way.

In the file `listfns.h`, you will see function headers for the five functions discussed in the previous section, except that this time the functions are written as global functions that accept `List` objects as parameters and do **NOT** have private access to the `List` implementation. Your task is to write a `listfns.C` file, which has the definitions of these five functions. The specifications are exactly the same as discussed in the previous section, except of course for the extra `List` parameter in each case.

Even though the specifications are the same, the requirements you have before you are slightly different. The global functions must have the same behavior as the member functions, but the global functions are allowed to use the `List` interface functions, as was just stated above. This was something you were not allowed to do for the first part, but you will *have* to do it for the second part, since you no longer have access to the private data of the `List` class nor do you have private access to its `ListNode` objects.

However, you are **NOT!!!!** allowed to use the functions you wrote in the first part to aid you in this part. That is, the two parts are distinctly separate, and for the purposes of this second part of the MP, you should assume that the five member functions you wrote for the first part of the MP do not exist. Once again, if you violate this rule, we will grade the relevant functions and output however we see fit.

5 Other tidbits

1. Make sure you understand the `List` ADT and its singly-linked implementation – i.e. the given code in `singlist.*` – before you even begin to design your algorithms for this MP. Otherwise, you will be quite lost, and taking random stabs at the problem isn't going to do you any good. Also, when you sit down to design the algorithms, think about them carefully, and take note of any special cases you need to worry about.

2. As of this writing, I am not aware of the status of Mattox's on-line PSP form, so I am including a Project Plan summary form to this handout. Your PSP task for this MP and for all remaining MPs is to turn in a filled out summary form. This means that you need to record your projections before you begin working on the MP, you need to record the relevant data as you work, and once you are done, you need to fill in the actual data for your work on the MP and make the relevant calculations for the data totals and quality measurements. Eventually, the on-line form will make the relevant calculations for you, and all you will need to do is record the data. But for now you need to do it all yourself. The official PSP assignment is to turn in a completed Project Plan summary form, but if the on-line form is completed before the MP is due, you can turn things in on-line instead. For now, assume no on-line form exists (meaning use the attached paper form), and if this changes during the week, we will let you know.

3. Please be aware that the "syntax grace period" of the grading policy ended with MP3. For this MP and all remaining MPs, you need to get the MP to compile to get any points, as we

now expect you are familar enough with the syntax of C++ that you won't get hung up for a week on something like a misdeclared pointer or a forgotten #include statement.

6 Handing in your code

To handin your MP4 code, use the command

```
handin cs225 mp4 singlist.C listfns.C
```

Project Plan Summary

Student _____ Date _____
Program _____ Program # _____
Instructor _____ Language _____

Summary	**Plan**	**Actual**	**To Date**
Minutes/LOC			
LOC/Hour			
Defects/KLOC			
Yield			
A/FR			
Program Size (LOC):			
Total New & Changed			
Maximum Size			
Minimum Size			

Time in Phase (min.)	**Plan**	**Actual**	**To Date**	**To Date %**
Planning				
Design				
Code				
Code Review				
Compile				
Test				
Postmortem				
Total				
Maximum Time				
Minimum Time				

Defects Injected	**Plan**	**Actual**	**To Date**	**To Date %**	**Def./Hour**
Planning					
Design					
Code					
Code Review					
Compile					
Test					
Total					

Defects Removed	**Plan**	**Actual**	**To Date**	**To Date %**	**Def./Hour**
Planning					
Design					
Code					
Code Review					
Compile					
Test					
Total					